Task-Driven Requirements in Object-Oriented Development

Article · January 2004

DOI: 10.1007/978-1-4615-0465-8_3

2 authors, including:

Barbara Paech
Universität Heidelberg
212 PUBLICATIONS **2,255** CITATIONS

SEE PROFILE

Chapter #

TASK-DRIVEN REQUIREMENTS IN OBJECT-ORIENTED DEVELOPMENT

Barbara Paech, Kirstin Kohler
Fraunhofer Institute for Experimental Software Engineering
Kaiserslautern, Germany
E-mail: {paech,kohler}@iesef.raunhofer.de

Abstract: There is no accepted method today that integrates requirements engineering and object-oriented development for user interface and information-intensive systems. In this paper we present the major issues such a method has to deal with and illustrate them with examples from our method TORE (Task and Object-oriented Requirements Engineering).

Key words: Requirements specification, Object-oriented development, User interface design, Tasks

1. INTRODUCTION

Object-oriented methods have penetrated software development in many application areas. As for the evolution of the structured methods, they focused first on code, but then gradually several notations for design and requirements were introduced, and were finally standardized with the unified modeling language (*UML*) [26]. At the same time, the unified process [15] was developed to standardize the application of these notations that widely differed before in methods such as [4,7,14,29]. For requirements engineering, the unified process - in industry mostly known in the specific form of the Rationale Unified Process (*RUP*) [19] – offers use cases and class diagrams.

While use cases were not part of the early object-oriented methods, they are now widely accepted as a good means to capture requirements. This is exemplified in the bulk of recent book publications on use cases, e.g. [1,6].

The most popular of these is [6], which treats use cases in isolation. It gives guidance on how to develop use cases on different levels of detail, but does not show how to integrate use case development into a full-fledged requirements engineering process. So, for example, little is said about how to integrate class modeling with use case modeling or how to integrate user interface development with use case modeling. Thus, despite the standardization efforts, there is no commonly accepted method today that integrates object-oriented development (*OO*) and requirements engineering (*RE*), not even for particular application domains.

Although there is no accepted method, it is possible to characterize the fundamental issues of such integration. It is the purpose of this paper to present and illustrate these issues for a particular domain, namely *user interface- and information-intensive systems* (in the following abbreviated as *UIS*). Object-oriented applications are typically from this domain. Examples are information or workflow systems to support business processes in a company, or web-applications for B2C. In our discussion we focus on *functional requirements*, since methods for non-functional requirements are just starting to evolve (e.g., [5]) and are not tailored to object-oriented methods.

The rest of this paper is structured as follows: in Sect. 2 we discuss the properties that a method integrating RE and OO should exhibit. Then we propose a conceptual model of functional UIS requirements that reflects these criteria. This model is used to classify two prominent, but quite different methods in this area. In order to illustrate the details of the integration, we sketch our own method *Task and Object-oriented Requirements Engineering (TORE)* in Sect. 3, using the example of a web-book store. Then we discuss how this method satisfies the criteria given in Sect. 2. We conclude with a summary and an outlook.

2. REQUIREMENTS SPECIFICATION METHODS FOR OBJECT-ORIENTED DEVELOPMENT

In this section we describe the essential concepts a specification method integrating RE and OO must support. In Sect. 2.1 we look at the stakeholders involved in system development and their needs wrt. specification. This uncovers four criteria that are refined into 16 concepts in Sect. 2.2. The resulting conceptual model covers the complete set of decisions that have to be made to specify UIS. We use the conceptual model in Sect. 2.3 to sketch how two prominent specification approaches differ in their support wrt. the elements of the conceptual model.

2.1 Integrating RE and OO for UIS

The purpose of requirements specification is to capture the information necessary for the stakeholders involved in system development so that they can efficiently contribute. Depending on their roles, these stakeholders have different needs:

- For procurers, product managers, and users, the specification must capture the value proposition of the procurers or managers and the needs of the users. This entails that it must be understandable by these stakeholders.
- For the system developers, the specification is the basis for the development. This means that it must be precise and consistent so that designers and implementers know what to build and quality assurance knows what to validate. In particular, this includes the user interface designers and usability testers.

In addition, the specification must support the project manager and the maintainers, but we are not concerned with these issues here.

Typically, there exists not one single specification that can serve both purposes. Thus, different notations and different views and abstraction levels are used. At some level functional requirements typically describe the input, output and behavior of system functions. However, there is little agreement on how to capture the system context, and what the exact boundary to design should look like.

Since we are dealing with OO, we stipulate that an analysis class model (and possibly some other preliminary OO models) should serve as the lowest level of requirements specification. The main difference to design class models is that they are not optimized wrt. design issues like cohesion and coupling and quality requirements such as performance.

Since we are dealing with UIS, we also stipulate that the requirements specification should explicitly state the user interface requirements. Note that usability is typically viewed as a non-functional requirement. However, the elements of the user interface are needed to realize system functions. That is why we treat them as functional requirements here. Thus, the main question is how to capture the system context.

Structured development methods start with a context diagram making the data flow between the environment and the system explicit, e.g. [12]. Based on this, high-level functions are described and decomposed. In later methods, these descriptions were complemented with data-oriented descriptions, such as entity relationship diagrams [11]. Early object-oriented methods start with a high-level class diagram capturing the domain relevant

for the development, and gradually evolve this to a class diagram of the system, e.g. [29].

RE methods often start with goals, where goals are high-level functional or non-functional requirements [5,6,20].

Human-computer interaction (*HCI*) methods typically start with the tasks of the users [9,13]. The tasks can be identified by looking at the current work of the users. The tasks are similar to the high-level activities identified by the structured methods through the context diagram, since both abstract from the actual system functions and focus on the context in which the functions are used. However, the viewpoint is different, since tasks emphasize the work context of the user, while the high-level activities emphasize the central role of the system to be built. Similarly, tasks are often covered by the goals identified in goal-oriented RE methods, but again the viewpoint is different. Goal-oriented RE methods start with the interests of the stakeholders, but not with the work context.

Considering these different ways to capture the system context, we stipulate for UIS that the specification process should start with tasks. Since UIS are developed to support work contexts, tasks must be made explicit in the UIS requirements.

Altogether, a method integrating RE and OO for UIS must satisfy the following criteria:

- For the procurers, product managers and users it must support the specification of the **system context**, in particular that of **user tasks**.
- For the OO developers it must support the specification of **analysis class diagrams**.
- For the user interface developers it must support the specification of **user interface requirements**.
- For all stakeholders it must support the specification of **system functions**.

In the following section we take a detailed look at the concepts behind these criteria.

2.2 A conceptual model for functional requirements of UIS

RE methods are typically characterized by the activities and notations they support. The activities lead to *decisions* that are documented with the notations (see also Kovitz [18]). During RE the stakeholders make decisions about the effect of the software system on the environment. Even if people who analyze and specify requirements do not think about this as a decision-making process , this is what they do: deciding about the behavior of the

system. These decisions constrain the solution space for the subsequent development activities (design and implementation). In the following, we will call them *requirements decisions* or just *decisions* for short.

Thus, more fundamentally than activities and notations, RE methods can be distinguished according to the following characteristics:

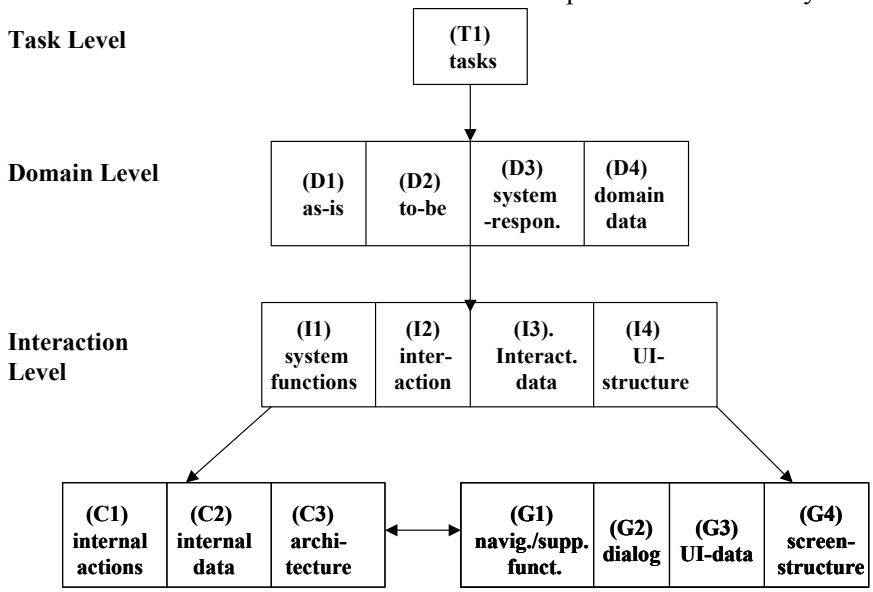
- **Decision types explicitly supported**: During RE, decisions have to be made. Approaches differ as to whether they offer support to explicitly make these decisions, or whether they leave them implicit. If decisions are made implicitly, this means they are made arbitrarily. Different people will make them differently.
- **Order of decisions**: Different approaches recommend a different order in the sequence of decisions. This has a big influence on the outcome of the decisions, because decisions made constrain the solution space for subsequent decisions.
- **Guidance to make decisions**: Approaches differ in the kind and amount of guidance they give to make decisions. This guidance helps to explore different options for the decisions and to make the right decision. Without explicit guidance, stakeholders are again at risk of making the decisions arbitrarily.
- **Documentation of decisions:** Notations are used to document decisions. Different approaches use different notations to document the same decision. The choice of a notation depends on the type of decision. Not every notation is suitable for documenting every type of decision. Sometimes, one notation can be used to document several decisions.

In the following, we present a conceptual model for the decision types that should be supported by methods integrating RE and OO. These decision types detail the criteria given in the previous section. In Sect. 2.3. we discuss which notations are used by two prominent methods to document these decisions.

As shown in Fig. 1, we identified 16 requirements decisions to be made for UIS, and aligned them on four abstraction levels:

- **Task level**: The motivation for users to use a UIS is their work. UIS support the tasks users perform as part of their work in a specific role. Decisions about the roles and tasks to be supported by the UIS are made on this level.
- **Domain level**: Looking at the tasks in more detail reveals the activities users have to perform as part of their work. These activities are influenced by organizational and environmental constraints. At this level, it is determined how the work process changes as a result of the new system. This includes, in particular, the decision on which activities will be supported by the system (called *system responsibilities*) and which domain data are relevant for these activities.

– **Interaction level:** On this level, decisions about the apportionment of activities between human and computer are made. They define how the user can use the system functions to achieve the system responsibilities. This decision has to be aligned with the decision about the UI structure, which the user can use to invoke the system functions.
– **System level**: Decisions about the internals of the application core and the graphical user interface *(GUI*) are on the system level. They determine details of the visual and internal representation of the system to be developed.



System Level: Application Core and GUI

Figure #-1. Decision Types

Each level corresponds to a specific view on the system and its context on a specific level of detail. Furthermore, the decisions on one level depend on the decisions of the previous levels. Decisions of one level have to be made after all decisions of the previous level have been determined. If decisions of lower levels are made without taking into account the higher-level decisions, the system will not adequately support the users in their tasks. Within one level decisions influence each other. The order between decisions of one level is arbitrary.

In the following, we explain the decision types represented in Fig. 1. In Sect. 3 we illustrate them with the example of a web-bookstore using the notations recommended in TORE.

- (T1) Decisions about the **user tasks**:
  The decisions determine the user roles and the tasks of these roles to be supported by the system. Business processes determine these tasks.
- (D1) Decisions about the **as-is** activities:
  The user tasks consist of several activities. As-is activities are the steps users currently perform as part of their work without the new system. Decisions must be made on what the as-is activities of a task are (as these are rarely explicit) and whether they are relevant for the system. These decisions shape the understanding of the purpose and the responsibilities of the new system.
- (D2) Decisions about the **to-be** activities:
  It needs to be decided how the as-is activities will change as a result of the new system. As-is activities always carry the potential for improvement. New technologies like the Internet or handheld devices can result in radically new to-be activities. To-be activities constitute the steps of the user tasks in the future.
- (D3) Decisions about the **system responsibilities**:
  Typically, the system does not support all to-be activities, but only a subset. These are the system responsibilities. These decisions clarify the key contribution of the system.
- (D4) Decisions about the **domain data** relevant for a task:
  System responsibilities of UIS manipulate data. Decisions have to be made on which domain data are relevant for the system responsibilities.
- (I1) Decisions about the **system functions**:
  System responsibilities are realized by system functions. The decision about the system functions determines the border between user and system.
- (I2) Decisions about user-system **interaction:**
  It has to be decided how the user can use the system function to achieve the system responsibilities. This determines the interaction between user and system.
- (I3) Decisions about **interaction data**:
  For each system function the input data provided by the user as well as the output data provided by the system have to be defined.
- (I4) Decisions about the structure of the user interface (**UI-structure):**
  Decisions about the grouping of data and system functions in different *workspaces* have to be made. System functions and data grouped in one workspace will be close together in the GUI. This means that users need less navigation effort in the interface to invoke system functions and

view data within the workspace. Through the UI-structure, the rough architecture of the user interface is defined. This structure has a big influence on the usability of the system.

- (C1) Decisions about the application **architecture**:
  The code realizing the system functions is modularized into different components. In the decision about the component architecture, existing components and physical constraints as well as quality constraints such as performance have to be taken into account. During requirements only a preliminary decision concerning the architecture is made. This is refined during design and implementation.
- (C2) Decisions about the **internal system actions**:
  Decisions have to be made regarding the internal system actions that realize the system functions. The system actions define the effects of the system function on the data. These decisions also define an order between the system actions as far as this is necessary to understand the behavior of the system function. In OO the system actions are grouped within classes. This is only a preliminary decision, which is refined during design and implementation.
- (C3) Decisions about **internal system data:**
  The internal system data refines the interaction data to the granularity of the system actions. The decisions about the internal system data reflect all system actions. In OO, system data is grouped within classes. Again, this is only a preliminary decision, which is refined during design and implementation.
- (G1) Decisions about **navigation** and **support functions:**
  It has to be decided how the user can navigate between different screens during the execution of system functions. This determines the navigation functions. In addition, support functions that facilitate the system functions have to be defined. These functions realize parts of system functions that are visible to the user, for example, by processing chunks of data given by system functions in a way that can be represented in the user interface. Another example are support functions that make the system more tolerant towards user mistakes.
- (G2) Decisions about **dialog** interaction:
  For each interaction the detailed control of the user has to be decided. This determines the dialog. It consists of a sequence of support and navigation function executions. These decisions also have a strong influence on the usability of the system.
- (G3) Decisions about detailed **UI-data:**
  For each navigation and support function, the input data provided by the user as well as the output data provided by the system have to be defined. These decisions determine the UI-data visible on each screen.

– (G4) Decisions about **screen structure:**
  The separation of workspaces as defined in (I4) into different screens that support the detailed dialog interaction as described in (G2) has to be decided. The screen structure groups navigation and support functions as well as UI-data. The decisions to separate the workspaces into different screens are influenced by the platform of the system.

The levels conform to a certain kind of pattern: At the domain level, the interaction level, the application core and the GUI, there are always decisions concerning behavior chunks like activities, functions or actions as well as decisions concerning data. Interaction and dialog put these chunks into a sequence. UI-structure, architecture and screen structure group data and behavior chunks together.

In order to ensure completeness of our conceptual model, we investigated methods integrating RE and OO wrt. these decisions types. In the following section we discuss two approaches in relationship to our model in more detail.

2.3 Comparing approaches with the conceptual model

For our discussion, we choose the use case approach by Armour/Miller [1], which details the RE approach of the RUP, and the Contextual Design approach by Beyer/Holtzblatt [3], which is an elaborate HCI approach. These two approaches can be viewed as two extremes: Armour/Miller emphasize embedding in a typical OO process and thus only use the notations from the UML. Beyer/Holtzblatt emphasize usability and use several new notations dedicated to task- and GUI-modeling.

Fig. 2 lists the decision types of the conceptual model in relationship to the notations used in the two approaches. “X” in a cell indicates that the notation, represented by that row, is used to document the decision represented in the column. If a decision column is empty, this means the approach does not support this decision type. Note that the approaches also differ wrt. guidance given on making decisions (see Sect 2.2), but this is not reflected in the table.

Armour/Miller cover most of the decision types of our conceptual model. One main characteristic of their approach is the continued usage of use cases at different levels. Use cases on a high abstraction level (initial use cases) are subsequently extended with additional fields to elaborated use cases. This continued usage of use cases minimizes the documentation effort, because models created in early steps of RE can be reused and extended during later steps. Armour/Miller provide very detailed guidance on how to extend and use the use cases at different levels. In addition to the different abstraction

levels, Armour/Miller also introduce “what-is” and “will-be” use cases. They support the documentation of our “as-is” and “to-be” decisions. Armour/Miller do not explicitly use system functions as part of their approach. The decisions about the system functions are hidden in the base and elaborated use cases and are only explicitly specified on the user interface level. Internal use cases describe the interaction of internal actions. Armour/Miller also distinguish between two types of data models: domain object model and analysis object model. The domain object model is used to document the domain-, data- and interaction-data decisions. The decisions about the UI-structure are not made explicitly, but left to the physical interface design.

	(T1) Task	(D1) As-Is	(D2) To-Be	(D3) System Respons.	(D4) Domian Data	(I1) System Function	(I2) Interaction	(I3) Interaction Data	(I4) UI-Structure	(C2) Int. System Actions	(C3) Internal Data	(C1) Architecture	(G1) Nav./Supp. Functions	(G2) Dialog	(G3) UI-Data	(G4) Screen Structure
Armour/Miller																
Use Case Diagramm				X												
Domain Object Modell (Glossary)					X			X								
Initial What-Is System Use Case		X														
Initial What Will Be System Use Case			X													
Base System Use Case							X									
White-Box Base System Use Case							X			X						
Elaborated System Use Case							X			X						
Internal Use Case										X						
Activity Diagram										X						
Architecture Document												X				
Analysis Sequence Diagrams										X						
Analysis Object Model											X					
Transaction Information Model						X									X	
Logical Screen Order														X		
Beyer/Holtzblatt																
Consolidated Work Model	X	X														
Storyboard			X	X			X									
Focus area						X		X					X			
UED									X							
Use Case																
Object Model																

Figure #-2. Decision Types and Notations of Armour/Miller and Beyer/Holtzblatt

Armour/Miller emphasize the importance of architecture decisions to balance the use case model. These decisions are documented in an

architecture document, which, however, is not described in detail. Whereas the decision types of the application core are covered by their approach, it neglects the user interface part. The user interface design is derived by turning the use cases into so-called *transaction information models*. A transaction is a system function call by the user and the system response as a result of the function together with the data involved. From each transaction a logical screen is derived. These logical screens are placed in sequence to form transaction trees. This order documents the dialog decision. Armour/Miller do not support the decisions about navigation and support functions nor the screen structure.

Contextual Design emphasizes the task, domain, interaction and user interface level. One of the notations invented by Beyer/Holtzblatt is the *work model*. It provides a very detailed picture of the tasks and as-is activities. As part of the work model, communication between the people involved, work artifacts, cultural and environmental constraints are documented. The to-be activities, system responsibilities and interaction are illustrated with the *storyboard*. In addition, Beyer/Holtzblatt introduce *focus areas* and the *User Environment Design* (*UED*) to document the UI-structure and screen structure. Focus areas document workspaces with their purpose, system functions, navigation functions, and interaction data. The complete set of focus areas builds the UED. In this approach some notations are used to document several decision, for example, the focus area and the storyboard. Whereas Contextual Design is very detailed in the description of the user interface, it omits the application core. Beyer/Holtzblatt mention that the focus area should lead to use cases and OO models that guide the implementation. The transition to these models is not described.

In addition to the two approaches described above, we also investigated several other approaches wrt. our conceptual model, e.g. [6,8,22,25]. We found that all decisions documented in these approaches were part of our conceptual model, but none of them covers all. Also, the order of the decisions and the guidance given as well as the notations used differed. Thus, we came up with our own approach to integrate RE and OO. It covers all decision types and thus exhibits all the issues of integrating RE and OO. This approach is sketched in the next section, and the decisions types are illustrated with an example.

3. TASK- AND OBJECT-ORIENTED DEVELOPMENT (TORE)

In this section we sketch our own method for integrating RE and OO for UIS: *Task- and Object-oriented Development* (*TORE*). The fundamentals of this method are described in [27]. An adaptation of this method for component-based product line engineering is described in [2]. In the following, we sketch the rationale for the development of TORE and the most prominent issues arising in the integration of RE and OO.

The most important driving factor for the development of TORE was to define a method that satisfies the criteria explained in Sect. 2.1. In particular, TORE was designed to support the identification and *explicit* specification of tasks, functions, analysis class diagrams, and user interface requirements.

	(T1) Task	(D1) As-Is	(D2) To-Be	(D3) System Respons.	(D4) Domian Data	(I1) System Function	(I2) Interaction	(I3) Interaction Data	(I4) UI-Structure	(C2) Int. System Actions	(C3) Internal Data	(C1) Architecture	(G1) Nav./Supp. Functions	(G2) Dialog	(G3) UI-Data	(G4) Screen Structure
TORE																
Business Process Diagram	X															
Role Description	X															
Task Description	X	X														
Activity Diagram and Activity Description		X	X	X												
Use Case Diagram				X												
Function Description						X										
Use Case Text							X									
ERD					X											
Glossary					X											
UI Structure Diagram									X							
Class Model								X		X	X					
Architecture Description												X				
Dialog Text														X		
Dialog Statechart														X		
Prototype													X		X	X

Figure #-3. Decision types and notations used in TORE

Another important driving factor in the design of TORE was to provide notations and guidance for *making all 16 requirements decisions explicit*. We are fully aware that in industrial application, there is rarely time to specify all decisions explicitly for the whole system, but we are convinced that depending on the context, different subsets of the decisions for different

subsystem parts should be specified explicitly. Thus, it is important to enable the developers to specify whatever they need [17].

According to the method characteristics discussed in Sect. 2.2., the last issue left is the question of which order the decisions should be made in. Fig. 1 shows the major dependencies between the decisions. TORE respects these dependencies and gives recommendations on the order within the levels. On each level, we use one or two models to drive the decisions, and the other models make the remaining decisions explicit, thereby consolidating the first decisions. This ensures consistency and completeness.

In the following, we show how to describe the decisions in TORE. We do not describe the TORE process completely, but only give hints on how the decisions of higher levels support decision making on the lower levels. We illustrate the specifications produced in TORE with the example of a web-bookstore. This should help the reader to understand the decision types in more detail. Fig. 3 gives an overview of the notations used in TORE for the different decision types. We discuss these notations in the following subsections, where each subsection describes the notations used for one of the four levels: task, domain, interaction and system. The latter is divided into core and GUI.

3.1 Task Level

Fig. 4 shows an activity diagram representing part of the business process of a bookstore.

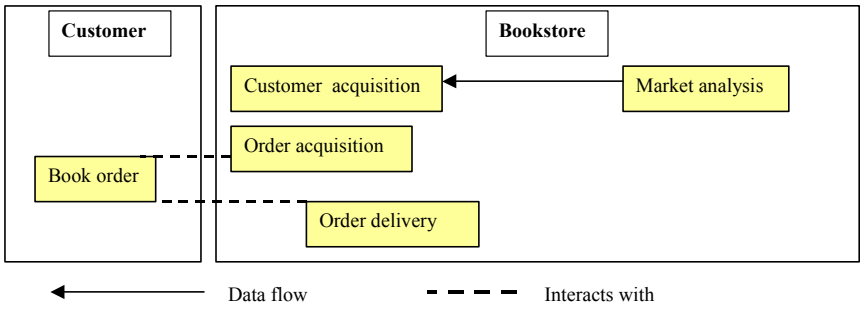


Figure #-4. Business Process

Starting with business processes is a good way to identify user tasks and roles (T1). Since UIS often radically change the ways business processes are conducted, it is important to make these changes explicit early on. Methods for creating business processes are described, e.g., in [16,30].

Table #-1. Role description

Role description `Customer`
Interests: The Customer wants to receive interesting books quickly and cheaply.
Tasks: The Customer is responsible for Book Order
Age/Gender: Adults (18-75 years old), male and female
Skills: Reads German, browses and searches the Iinternet, no other specific IT skills
Environment: Email access, browser Netscape or Explorer, average PC equipment

Parallel to the process descriptions, role descriptions are developed to capture the interests and responsibilities of the future system users. This is enhanced with description of skill levels that are needed for usability considerations. Table 1 shows such a role description for the `customer`.

The main purpose of business process modeling in TORE is to identify the tasks. Thus, for each role one identifies the relevant tasks in each business process and creates task descriptions for them. As an example consider Table 2, which shows a task description for `Book order`. The task description typically mentions the as-is activities implicitly.

Table #-2. Task description

Task description `Book order`
Description: Within this task the customer selects books from the bookstore. The bookstore gets the money from the customer. The customer receives the selected books from the bookstore
Performance: This task will be carried out 10,000 times a day by different customers worldwide.
Frequency: The average user will carry out this task between once per month and once per year.
Trigger: no special trigger, whenever customer likes
Risks: customer pays, but does not receive books

3.2 Domain Level

Based on interviews or work observations (or other elicitation methods, see [23,10]), the user tasks are refined with further activity diagrams to identify the system responsibilities.

(D1) As-is activities

First, the as-is activities are identified. As-is activities are often only described textually in the task description. Sometimes, however, it is important to explore the as-is activities in more detail. Then further activity diagrams can be drawn, and activity descriptions similar to the task descriptions are created. In both cases, problems with the as-is activities mentioned by the users have to be captured and need to be taken care of during development.

(D2) To-be activities

Based on the as-is activities, the to-be activities are defined. Again, this can be done textually or with diagrams. Here, IT experts need to discuss with the users major changes in the existing work processes induced by new IT possibilities. One major difficulty is to know when to stop refinement of the tasks. Since the activity descriptions serve to identify the system responsibilities, we recommend stopping as soon as an activity can clearly be associated with the system. It is a matter of the interaction level to decide in which ways these activities are supported by the system.

In the example, as-is activities detail how the user buys books in a conventional book store. The to-be activities detail how the user orders books from an Internet bookstore, that is: the `customer` has to `Select Books`, to `Provide Customer Data` and to `Place Order`. Both are supported by the system. Thus, they are system responsibilities.

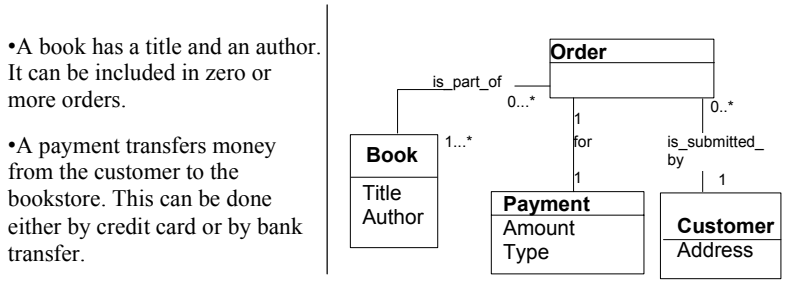


Figure #-5. Domain data

(D3) Domain data

To consolidate the activity models, a data model is developed in parallel, which has to cover all the data mentioned in the activity descriptions. This can be a glossary for the major terms used in the task descriptions. It can also capture data structure in terms of an entity relationship diagram (*ERD*). In TORE we do both, since ERDs are the first step on the way to an analysis class model. In addition to the ERD, each class is described textually so that these descriptions serve as glossary entries.

Fig. 5 shows part of the data model of the bookstore. At this level, only class name, associations and possibly attributes are captured to characterize the data associated with a class.

(D4) System responsibilities

The system responsibilities are collected in a list structured according to the different roles involved in these activities. To facilitate discussion, this

should be visualized in a use case diagram as in Fig. 6. We use the notation adapted from [21], where the system border crosses the use case bubbles. This highlights the fact that these activities are supported by the system. The exact border between the human and the system has to be determined on the interaction level.

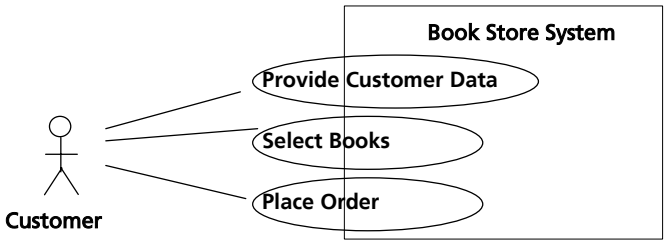


Figure #-6. Use case diagram

3.3 Interaction Level

The adequate border between human and system can only be defined in close interaction with the users. There are several models that can support discussion with the users. It can either focus on the interaction or on the UI-structure, or on the interaction data. In TORE we recommend using one of the first two, since there is evidence that users have difficulties discussing abstract data models [24]. The reason is that class models abstract too much from the representations users are accustomed to in their work. The system functions are described after the UI-structure has been decided.

(I2) Interaction

The discussion of interactions to determine the border is based on a use case description for each system responsibility. As shown in Table 3, this description explicitly names system responsibilities. We employ a use case template adapted from [6, 8]. From the latter we take the explicit distinction between actor and system activities. The template facets are similar to the ones of the RUP. One difference is our emphasis on locating exceptions in the description of the flow of events. In our experience, a detailed analysis of exceptions is a prerequisite for a complete understanding of the dynamics.

The main problem with use case descriptions is the abstraction level. Since we use them on the interaction level, we do not describe user interface details and abstract from screens, UI-data, and navigation and support functions, although the latter can often be found in practice. Instead, the use cases show for each system responsibility how the user navigates between workspaces.

(I4) UI-structure

Another way to determine the border between humans and system is to discuss the UI-structure with the users. As shown in Fig. 7, this structure groups data and system functions together similar to the workspaces of [3].

At first, the system responsibilities are shown in this structure, and then they are gradually replaced with the identified system functions. Here the user's imagination is not focused on the dynamics, but more on the question of which information is presented in which context. This can also be supported by UI-prototypes. However, such prototypes bear the risk of users concentrating on UI-details, instead of on the major structure.

Table #-3. Use case text for Place Order

Name Place Order
Realized User Task Book Order
Initiating Actor Customer
Participating Actor Bookstore Clerk
Flow of events 1. The System displays the shopping basket with the selected book. 2. The Actor selects the "Place Order"-responsibility. [No Customer Data] 3. The System shows order and shopping basket and supports the Actor in determining the payment method and the address and submitting the order. [New selection] [New customer data] [No order] 4. The System acknowledges the order to the Actor, stores the order and supports the Clerk with the "Order Delivery"-responsibility. 5. The Actor receives the selected books
Exceptions [No Customer Data] The System does not have information on the address and payment methods of the Actor. The System changes to the "Provide Customer Data"-Responsibility. When this is successfully finished, the System continues with 3. [New selection] The Actor decides to change the shopping basket and selects the "Select Book"-Responsibility. The System preserves the data submitted so far and changes to the "Select Book" Responsibility. After successful completion of selection, the System continues with 3. [New customer data] The Actor decides to change his or her data and selects the "Provide Customer Data"-Responsibility. When this is successfully finished, the System continues with 3. [No order] The Actor does not submit an order before leaving the bookstore system. The System stores the data submitted so far. When the Actor revisits the bookstore system, the data will be shown to him.
Precondition Shopping basket is not empty
Postcondition Actor has ordered and received books

Name Place Order
Rules Payment must be either by credit card or by bank transfer
Quality Requirements Security/Privacy: Data about payment transaction and customer has to be protected

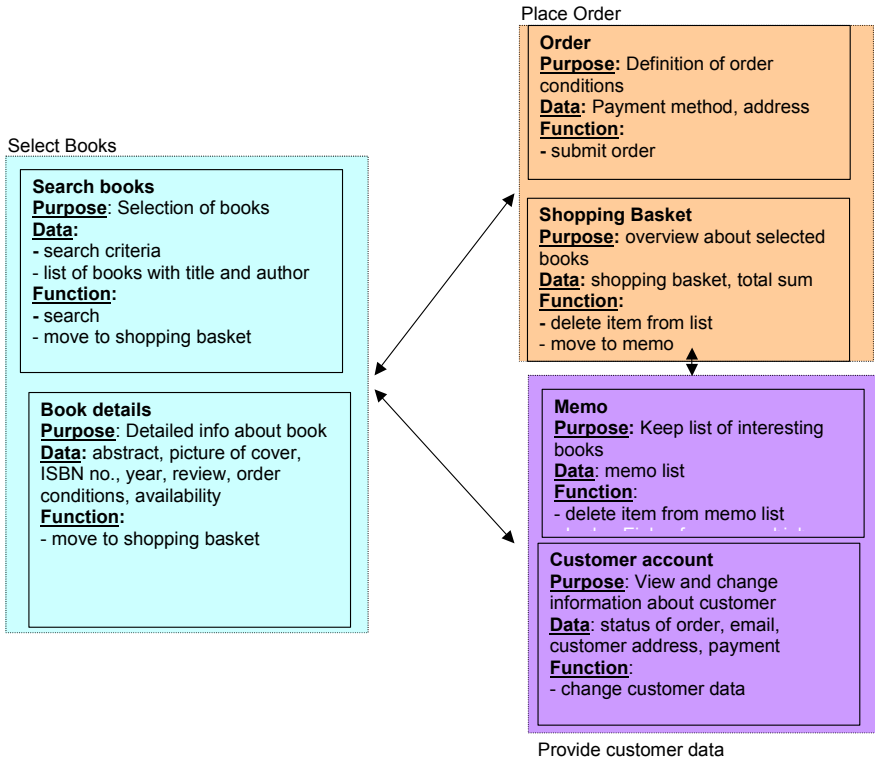


Figure #-7. UI-structure

(I3) Interaction Data

The data model from the domain level is refined at the interaction level. This model is used to consolidate the decisions made during use case or UI-structure elaboration. In the example, the data model is extended with a shopping basket class detailing the association between books and order. We do not yet define methods at this stage, since they are not visible to the user and therefore not important at the interaction level.

(I1) System functions

In parallel, for the more complex system functions, a function description is developed that captures the data changes. The template is similar to the function descriptions of FUSION [7]. In particular, they explicitly name the data input and output of the function. This allows for easy cross-checks with the data model.

3.4 System Level: Application Core

At this level in TORE, the OOA model is developed fully. This step can be carried out in parallel with the GUI development, before or after it. The results have to be synchronized.

(C3) Architecture

At first, a preliminary architecture specification is developed to capture physical constraints given by the customer and decisions necessary to refine non-functional requirements and functional requirements in parallel (e.g., in order to specify security, the architecture has to be known).

(C1) Internal actions and (C2) internal data

In TORE, decisions about the code component are documented as they are in the KOBRA method [2]. The essence of KOBRA is a recursive process of specification and realization. Fig.8 shows the description techniques involved.

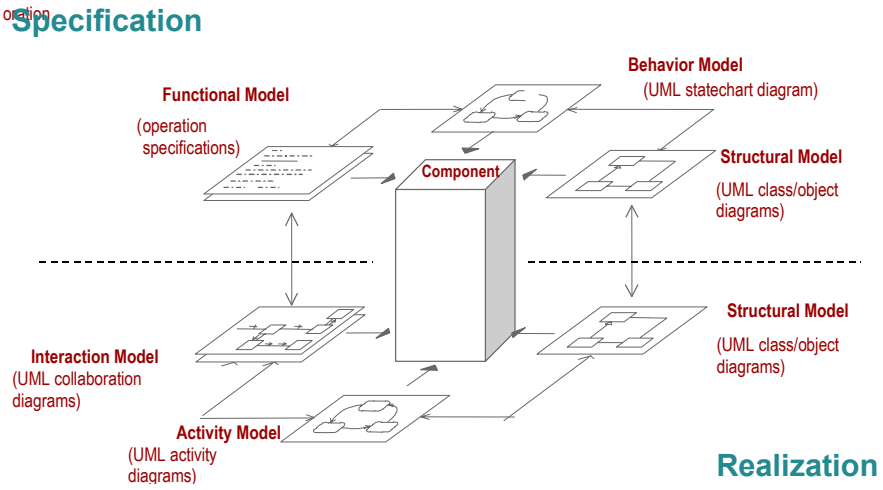


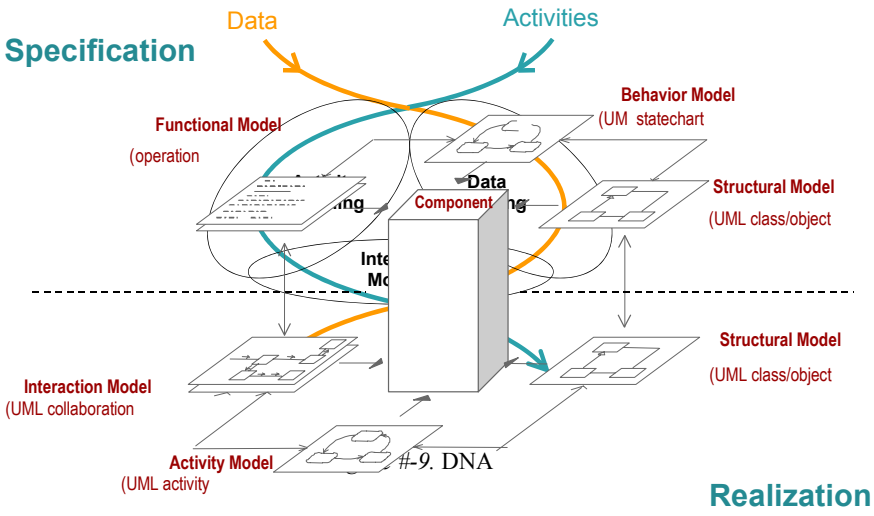
Figure #-8. KOBRA development

In TORE, the only iteration of the KOBRA process that is important is the top iteration , which describes how the system functions are realized through the interaction of analysis classes. Further iterations are design iterations.

The function descriptions and the class model constitute the specification. The use cases describe the context of the system functions and thus substitute the behavior model of the system used in KOBRA.

The purpose of the realization is to capture the details of the system functions in terms of interaction between analysis classes. That is, the internal actions are the methods of the classes, and the internal data are the classes and their attributes. The major issue is how to distribute the business logic realized in one function to the different classes. To avoid premature distribution resulting in complex class interactions, in KOBRA and TORE we use the *Data aNd Activity* (*DNA*) approach.

Fig. 9 illustrates the essence of the DNA approach: starting with the function descriptions, the actions necessary to realize the function are explored in parallel with an activity diagram, and the data input, modified and output through the function is refined into a class model (without methods). Then the identified actions are allocated to the identified classes, resulting in a full-fledged class model.



#-9. DNA

In the example, the activity model for the `Submit Order`- function is trivial, It only consists of creating a shopping basket, a payment method, an address, and an order. These activities can be straightforwardly associated with the classes given so far. The only choice is where to locate the control of the function, that is, who calls the methods to create the `shopping basket`, `payment method`, `address` and `order`. According to

[14], one could define a new class `order function` with one method responsible for coordinating the other methods. Another straightforward choice is to give the control to the `order` class, so that the method `Create order` creates all the other objects. However, since an order is not created if the user cancels the order activity, we prefer to define an order function class.

Since in TORE, function descriptions are created only for complex system functions, the class model gained in realization is not complete. Completion can be left to design. Alternatively, further methods are defined by exploring the classes in detail as described in typical OOA methods. Furthermore, the decisions on the GUI induce support functions and thus, new methods, and sometimes also classes.

3.5 System level: GUI

At this level in TORE the details of the user interface requirements are decided.

(G4) Screen structure and (G1) navigation/support functions and (G3) UI-data

The bulk of the work here consists of refining the UI-structure into a screen structure for a specific platform. Criteria for the mapping of the workspaces onto an adequate set of screens are described in [22]. The navigation between these screens is detailed with navigation functions. When designing the navigation of this screen structure, it is important to watch out for subtle dependencies between UI-steps and system actions defined in the application core. Sometimes, support functions have to be designed, providing part of the result of a system function. In addition, the details of the presentation of the data on the screen have to be determined. This includes the selection of appropriate user interface elements (e.g., is the user offered a choice or is he or she required to type in some text).

In TORE, we use GUI prototyping tools to define the screen structure. A GUI prototype allows to capture three decisions in one model: screen structure, navigation functions, and the UI-data. One can choose between high-fidelity prototypes (e.g., linked HTML pages without functionality) or low-fidelity prototypes (e.g., pencil drawings on paper). Low-fidelity prototypes have the advantage that they are easy to change. High-fidelity prototypes can be used to perform usability tests first and thus, to get early user feedback. For a more detailed discussion about high versus low fidelity prototypes, see also [28]. If high-fidelity prototypes are used, then the decision about the dialog can also be captured as part of the prototype.

(G2) Dialog

The dialog refines the interaction. It defines the dynamic sequencing of the screens. To get an overview of complex screen dependencies, state diagrams are used to describe the dialogs.

Fig. 10 shows part of a dialog for the `Place order` use case. It involves screens from the `Place Order` Responsibility and the `Provide Customer Data` Responsibility. The states are labeled with the data shown or with the name of the screen the transitions are labeled with navigation functions. The system functions (including support functions) called are shown as the second part of the transition label. Thus, in the example, two navigation functions *complete order* and *submit order* are introduced. Note that in general, the dialogs of different use cases should be combined into one state diagram to get a complete overview about the dynamics of a set of screens.

Another possibility for detailing a dialog is to enrich the use case text with details about screen structure and navigation functions. This is especially helpful for an in-depth discussion of a particular dialog. It does not, however, provide a complete picture of the different screens.

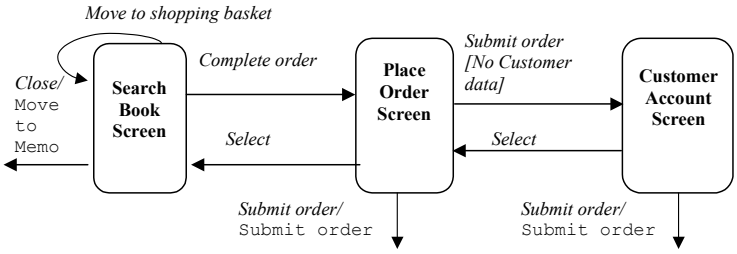


Figure #-10. Dialog

3.6 Experiences with TORE

Since 1998 we have been using TORE to teach students and practitioners the fundamentals of task-driven requirements specification. We have also performed major case studies, e.g., in the context of KOBRA. Parts of this have been applied in industrial development, e.g., for a logistic systems. Our experience is that it takes some time until computer scientists understand the value of a task-driven approach. Typically, they like to start directly from a class model or from system functions. However, it turns out that the discussion about the task structure is very helpful for creating a common

understanding of the system to be developed. Thus, after some time the software engineers appreciate having a clear basis for function definition. The main advantage for the customers is that this way, only the necessary system functions are implemented. When tasks are used to drive release planning, another advantage is that releases completely support tasks.

4. CONCLUSION

We have presented the fundamental issues in integrating requirements engineering with object-oriented development for user interface- and information-intensive systems. As UIS focus on supporting work contexts, the requirements process must be driven by tasks, and user interface requirements must be specified explicitly. We have identified 16 decision types covering the result of the RE process for UIS. The alignment of these decision types on the task, domain, interaction and system levels shows how task and user interface decisions complement the activity- and data-oriented decisions typically covered by RE and OO methods. A comparison with a typical OO method and a typical HCI method for UIS shows that only part of these decision types are covered by existing methods. This is due to the fact that the software engineering and the human-computer interaction communities are quite far apart nowadays. RE can serve as a bridge between them. We have to make software engineers, human-computer interaction experts in general, and requirements engineers in particular, aware of all decision types. As illustrated with the TORE method, this does not require new notations or new tools. Thus, the main obstacle for industrial acceptance of such an integrated method is the mindset of the people. A good way – if not the only one – to change this is to gather empirical evidence about the effectiveness of the integrated methods.

ACKNOWLEDGEMENT

We thank the RUE department at IESE, Soren Lauesen, und the editors for helpful comments on an earlier version of this paper. This work was funded by the BMBF in the EQF project under the label: e-Qualification Framework - VFG0008A

REFERENCES

1. Armour, F. ,Miller, G., *Advanced Use Case Modeling*, Addison-Wesley, 2000

2. Atkinson,C., Bayer, J., Bunse, Ch., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J., *Component-based product Line Engineering with UML*, Addison Wesley, Component Software Series, 2002
3. Beyer, H., Holtzblatt, K., *Contextual Design: Defining Customer Centered Systems*, Morgan Kaufmann Publishers, 1998
4. Booch, G., *Object-Oriented Analysis and Design with Applications*, Benjamin Cummings, 1994.
5. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J., *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 2000.
6. Cockburn, A., *Writing Effective Use Cases*, Addison Wesley, 2001
7. Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist, H., Hayes F., and Jeremaes P., *Object-oriented Development: The Fusion Method*, Prentice Hall, 1994.
8. Constantine, L., Lockwood, L., *Software For Use*, Addison Wesley, 1999
9. Diaper, D., *Task analysis for human-computer interaction*. Ellies Horwood, 1989
10. Dray, S. & Mrazek, D. "A day in the life:" Studying context across cultures. In J. Nielsen & E. del Galdo, Eds. *International User Interfaces*, John Wiley & Sons, 1996.
11. Downs, E., Clare, P., Coe, I., *Structured Systems Analysis and Design Method: Application and Context*, Prentice Hall, 1992
12. DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, 1978.
13. Hackos, J.T., Redish, J.C., *User and Task Analysis for Interface Design, John Wiley & Sons, 1998*
14. Jacobson I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
15. Jacobson, I, Booch G., Rumbaugh J., *The Unified Software Development Process,* Addison Wesley, 1999
16. Jacobson, I., Ericsson, M., Jacobson, A., *The Object Advantage: Business Process Reengineering with Object Technology*, Addison-Wesley, 1994
17. Kohler, K., Paech, B., "Requirement Documents that Win the Race, Not Overweight or Emaciated but Powerful and in Shape", *First Workshop for Time Constrained Requirements Engineering TCRE'02*, 2002
18. Kovitz, B.L. *Practical Software Requirements. A Manual of Content and Style*, Greenwich: Manning Publications Co., 1998
19. Kruchten, P. B., *The Rational Unified Process: An Introduction*, Addison-Wesley, 2000.
20. van Lamsweerde, A., Darimont, R., Massonet, P. (1998): "Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learned," *Int. Symp. on Requirements Engineering*, 194–203
21. Lauesen, S., *Software Requirements – Styles and Techniques,* Addison Wesley, 2002
22. Lauesen, S., Harning, S., "Virtual Windows: Linking User Tasks, Data Models and Interface Design*", IEEE Software*, pp. 67-75, July/August 2001
23. Macaulay, L.A., *Requirements Engineering*, Applied Computing, Springer Verlag, 1995
24. Moynihan, T., "Objects versus Functions in User-Validation of Requirements: Which Paradigm Works Best?, OOIS, pp.54-73, 1994
25. Oestereich, B., *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language*, Oldenbourg Wissenschaftsverlag GmbH, 2001
26. OMG, " The Unified Modeling Language", http://www.omg.org/uml/
27. Paech, B., *Aufgabenorientierte Softwareentwicklung- Integrierte Gestaltung von Unternehmen, Arbeit und Software*, Springer Verlag, 2000

28. Rudd, J., Stern, K., Isensee, S., Low vs. High Fidelity Prototyping Debate, *Interactions*, Vol.2, No. 1, 1996
29. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall International, 1991
30. Scheer, A, *ARIS – Business Process Frameworks, Business Process Modeling*, Springer-Verlag, 1999